Constructing Induction Rules for Deductive Synthesis Proofs

Constructing Induction Rules for Deductive Synthesis Proofs *

Alan Bundy Jeremy Gow Jacques Fleuriot Lucas Dixon

February 16, 2005

Abstract

We describe novel computational techniques for constructing induction rules for deductive synthesis proofs. Deductive synthesis holds out the promise of automated construction of correct computer programs from specifications of their desired behaviour. Synthesis of programs with iteration or recursion requires inductive proof, but standard techniques for the construction of appropriate induction rules are restricted to recycling the recursive structure of the specifications. What is needed is induction rule construction techniques that can introduce novel recursive structures. We show that a combination of rippling and the use of meta-variables as a least-commitment device can provide such novelty.

1 Introduction

One of the most under-exploited techniques in the arsenal of formal methods of system development is the deductive synthesis of programs from a constructive proof of their specifications. Deductive synthesis presents many difficult technical challenges, and this may be one reason for its relative neglect. In this paper we address one such challenge: the choice of induction rules in the presence of existential quantifiers. We describe some new techniques for the automatic construction of induction rules that provide an approach to this problem.

2 Deductive Synthesis

For expository purposes, we will adopt an especially simple version of deductive synthesis. This is illustrated in Figure 1. Programs will be represented as recursive functions and specifications as formulae within the same higher-order, typed, constructive logic. This will enable us to finesse issues of program semantics and to turn synthesis conjectures into verification conjectures by substituting synthesised programs for existential variables, as in Figure 1.

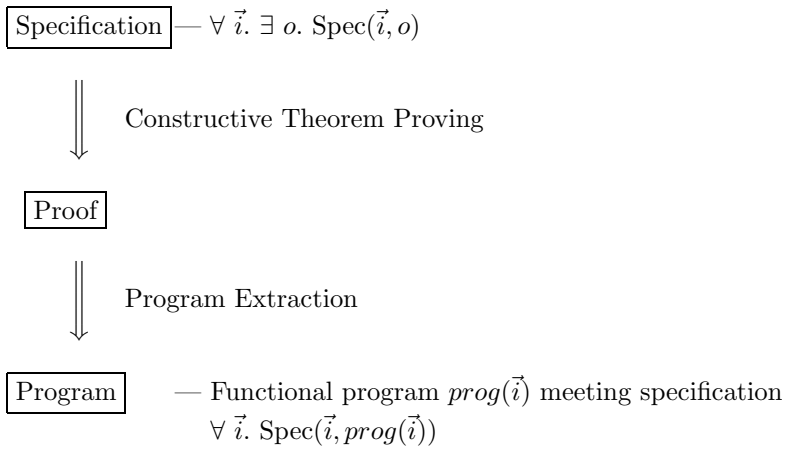
To illustrate the process, consider the task of synthesising a sorting algorithm $sort : list(\mathbb{N}) \mapsto list(\mathbb{N})$, i.e. a function whose input is a list of natural numbers and whose output is an ordered permutation of the input list. The synthesis conjecture might be:

$$\forall l{:}list(\mathbb{N}).\exists m{:}list(\mathbb{N}).\ ord(m) \wedge perm(l, m)$$

where $ord : list(\mathbb{N}) \mapsto bool$ is a predicate for testing whether a list is ordered and $perm : list(\mathbb{N}) \times list(\mathbb{N}) \mapsto bool$ is a predicate for testing whether one list is a permutation of another. We adopt the convention that lower-case roman letters stand for object-level variables or constants, whereas upper-case roman letters stand for meta-variables that range over object-level expressions.

The proof will construct a witness for the existential variable m. This witness will be a function of l, which we will call $sort(l)$. Since the logic is constructive, *sort* will be (recursively)

*The research reported in this paper was supported by EPSRC grant GR/S01771 for the first and third author and an EPSRC studentship to the second author. We are grateful for feedback from Andy Fugard.



Spec($\vec{i}, o$) is a logical specification of the relationship between the inputs $\vec{i}$ to the required program and its output o. The conjecture to be proved is that whatever the input there is a always an output that meets this specification. Constructive proof is used to ensure that a suitable output, $prog(\vec{i})$ is constructed as a side effect of the proof. This output provides the definition of the required program. By construction, this program is known to meet its specification. Some steps of the proof provide program operations. For instance, case splits provide conditional branches and induction steps provide recursive definitions.

Figure 1: Deductive Synthesis from Constructive Proof

defined in terms of previously defined functions. The proof will have verified that $sort(l)$ meets its specification, i.e. that:

$$\forall l{:}list(\mathbb{N}).\ ord(sort(l)) \wedge perm(l, sort(l)).$$

The kind of sorting algorithm that is synthesised will depend on the details of the proof [Darlington, 1978].

3 Induction Rules

Figure 1 notes the correspondence between the steps of a synthesis proof and the steps of the program they synthesise. In particular, applications of induction rules in the proof insert recursion in the synthesised program. Moreover, the kind of induction rule determines the kind of recursion. In imperative programs, inductive steps will create iteration or loops. More generally, induction is needed whenever some form of repetition is required in the synthesised object. Repetition arises in recursive data-structures, recursive or iterative programs, temporal change, parameterized hardware, etc., i.e. in nearly all non-trivial systems. Induction is, thus, of central importance in deductive synthesis.

There are many recursive data-types: natural numbers, integers, rationals, lists, trees, sets, etc. For each recursive data-types there are infinitely many induction rules. They can all be derived from the general schema of noetherian induction (also known as well-founded induction).

$$\frac{\forall x{:}\tau.\ (\forall y{:}\tau.\ y \prec x \rightarrow \phi(y)) \rightarrow \phi(x)}{\forall x{:}\tau.\ \phi(x)} \quad (1)$$

where $\prec$ is some well-founded relation on the type τ. By *well-founded* we mean that there are no infinite, descending chains of the form $\ldots \prec a_3 \prec a_2 \prec a_1$. The infinitely many possible well-founded relations $\prec$ for each non-trivial data-type τ give infinitely many possible instantiations

of this noetherian schema. Since it is not possible to pre-store all well-founded relations $\prec$ on all types τ, most inductive theorem provers construct induction rules on demand. The universally quantified variable x is called the *induction variable*. It is also possible to simultaneously induce on more than one variable, but in the interests of simplicity we omit this additional complexity here, but will return to it below.

The practical situation is more complex than this. The noetherian schema is rarely used directly. Usually, we use an induction rule derived from it, such as the following rule for the type $list(\tau)$.

$$\frac{\phi[\,] \quad \forall l{:}list(\tau).\ l \neq [\,] \wedge \phi(tl(l)) \rightarrow \phi(l)}{\forall l{:}list(\tau).\ \phi(l)} \tag{2}$$

where $[\,]$ is the empty list and $tl(l)$ is the tail of the list l. This induction rule is based on some well-founded relation $\prec$ under which $tl(l) \prec l$. Many such relations, *prec*, will suffice, for instance, one based on the size of the list. The first premise of this rule, $\phi[\,]$, is an example of a *base case*; the second premise, $\forall l{:}list(\tau).\ l \neq [\,] \wedge \phi(tl(l)) \rightarrow \phi(l)$, is an example of a *step case*. The antecedent of the step case, $\phi(tl(l))$, is the *induction hypothesis* and the consequent, $\phi(l)$, is the *induction conclusion*. The function $tl(l)$ is an example of a *destructor function*, as it destructs the recursive data-type. When a destructor function surrounds the induction variable in the induction hypothesis, we say that the induction rule is in the *destructor-style*. To formulate a destructor-style induction rule it is necessary to identify the base and step cases and to find a well-founded relation under which the destructor functions output strictly smaller terms than their inputs.

An alternative to destructor-style induction rules is *constructor-style*. In constructor-style rules the destructor function in the induction hypothesis is replaced by a constructor function in the induction conclusion. A *constructor function* constructs new elements of the recursive data-type from old. For instance, the constructor function $[h|t]$ constructs a new member of the type $list(\tau)$ from a list $t{:}list(\tau)$ and an element $h{:}\tau$. The constructor-style induction rule corresponding to the destructor induction (2) is:

$$\frac{\phi([\,]) \quad \forall h{:}\tau.\forall t{:}list(\tau).\ \phi(t) \rightarrow \phi([h|t])}{\forall l{:}list(\tau).\ \phi(l)} \tag{3}$$

where $[\,]$ is the empty list and $[h|t]$ is the list constructed by putting h at the head of the list t. The base case of this rule is also $\phi([\,])$ and the step case is $\forall h{:}\tau.\forall t{:}list(\tau).\ \phi(t) \rightarrow \phi([h|t])$.

As we will see below, it is also possible to have hybrid destructor/constructor-style induction rules. It is also possible to have inductions on more than one induction variable, such as the following constructor-style rule for the data-type of binary trees.

$$\frac{\phi(leaf(e)) \quad \forall l{:}tree(\tau).\forall r{:}tree(\tau).\ \phi(l) \wedge \phi(r) \rightarrow \phi(node(l,r))}{\forall t{:}tree(\tau).\ \phi(t)}$$

where $leaf(e)$ constructs a leaf of the tree with label e and $node(l,r)$ constructs a new binary tree from the left and right subtrees l and r.

The rules above are all, so called, *structural induction rules*, i.e. they have used the destructor and constructor functions from the recursive definition of the data-type. Non-structural rules are also possible, for instance,

$$\frac{\phi([\,]) \quad \forall l{:}list(\tau).\ \phi(butlast(l)) \rightarrow \phi(l)}{\forall l{:}list(\tau).\ \phi(l)} \tag{4}$$

where $butlast(l)$ outputs the list l with the last element deleted.

4 Synthesising Recursive Programs

Suppose induction is used to prove a synthesis conjecture, such as

$$\forall l{:}list(\tau).\exists m{:}list(\tau).\ spec(l,m)$$

A program synthesised using the destructor-style, induction rule (2), with l as the induction variable, will have the following recursive form.

$$\begin{array}{rcl} prog(l) & = & \text{if } l = [\,] \text{ then } b \\ & & \text{else } f(l, prog(t(l))) \end{array}$$

where b and f do not contain $prog$. On the other hand, a program synthesised using the constructor-style, induction rule (3) will have the following recursive form.

$$\begin{array}{rcl} prog([\,]) & = & b \\ prog([h|t]) & = & f(h, t, prog(t)) \end{array}$$

And a program synthesised using the non-structural, induction rule (4) will have the following recursive form.

$$\begin{array}{rcl} prog(l) & = & \text{if } l = [\,] \text{ then } b \\ & & \text{else } f(l, prog(butlast(l))) \end{array}$$

To synthesise a pair of mutually recursive programs, we would have two existential variables:

$$\forall l{:}list(\tau).\exists m_1{:}list(\tau), \exists m_1{:}list(\tau).\ spec(l, m_1, m_2)$$

from which we would extract $prog_1$ and $prog_2$ as the existential witnesses of m_1 and m_2, respectively. If constructor-style, induction rule (3) were used to prove this synthesis conjecture, then the following mutually recursive definitions would be synthesised.

$$\begin{array}{rclrcl} prog_1([\,]) & = & b_1 & prog_2([\,]) & = & b_2 \\ prog_1([h|t]) & = & f_1(h, t, prog_1(t), prog_2(t)) & prog_2([h|t]) & = & f_2(h, t, prog_1(t), prog_2(t)) \end{array}$$

5 Constructing Induction Rules

Recursion analysis is the best known technique for constructing customised induction rules for specific conjectures. It is due to Boyer and Moore and was implemented in their Nqthm prover [Boyer & Moore, 1979]. The essential idea is to identify recursively defined functions in the conjecture and then convert these into the corresponding induction rules. For instance, if the conjecture contained a function g whose recursive definition was $g(k, l) = f(k, l, g(k, tl(l)))$, then the destructor induction rule (2) with induction variable l would be suggested. Boyer and Moore developed techniques for merging and generalising the suggestions from the different recursive function into one induction rule that subsumed them all. Walther later refined and improved these rule merging techniques [Walther, 1993]. The heuristic underlying recursion analysis is that by choosing an induction hypothesis containing the same destructor functions as the recursive definitions, we maximise the chances that these definitions will be able to manipulate the hypothesis. Recursion analysis can also be adapted to constructor style definitions and induction rules. Recursion analysis is illustrated in Figure 2.

Recursion analysis was developed for purely universally quantified conjectures. When it comes to conjectures containing existential quantifiers, especially the conjectures used in deductive synthesis, it suffers from a major drawback.

- The induction rule used in the proof will determine the recursive structure of the synthesised program, i.e. its fundamental algorithmic nature, including its complexity.
- Recursion analysis will choose this induction rule using the forms of recursion it finds in the conjecture, i.e. the algorithmic nature of the *specification*.
- Thus the programs constructed by deductive synthesis are algorithmically similar to their specifications.

Conjecture: $\forall l{:}list(\tau).\exists m{:}list(\tau).\ perm(l,m)$

Recursive Definition:

$$\begin{array}{rcl} perm(l,m) & = & \text{if } l = [\,] \text{ then } m = [\,] \\ & & \text{else } perm(tl(l), del(hd(l), m)) \end{array}$$

where $del(e,l)$ deletes the element e from the list l.

Constructed Induction Rule:

$$\frac{\phi[\,] \quad \forall l{:}list(\tau).\ l \neq [\,] \wedge \phi(tl(l)) \rightarrow \phi(l)}{\forall l{:}list(\tau).\ \phi(l)}$$

perm is the only recursively defined function that appears in the conjecture. Its recursive definition is on l, the first argument of perm, and the recursive call on this argument is tl(l). This suggests constructing a one-step, destructor-style induction in which l is the induction variable and the induction hypothesis is applied to tl(l). When this induction rule is applied, the induction term, $\phi(l)$, will be instantiated to $\exists m{:}list(\tau).\ perm(l,m)$.

Figure 2: An Example of Recursion Analysis

This is not a desirable state of affairs. For instance, it means that recursion analysis is unable to construct the induction rule needed for the synthesis of quick-sort, because its recursive structure is radically different from that of either *ord* or *perm*. We will use the definition of *perm* in Figure 2 and the following definition of *ord*:

$$\begin{array}{rcl} ord(l) & \leftrightarrow & \text{if } l = [\,] \text{ then } \top \\ & & \text{elseif } l = [h] \text{ then } \top \\ & & \text{elseif } hd(l) \leq hd(tl(l)) \text{ then } ord(tl(l)) \\ & & \text{else } \bot \end{array}$$

Whereas the usual definition of quick-sort is:

$$\begin{array}{rcl} qsort(l) & = & \text{if } l = [\,] \text{ then } [\,] \\ & & \text{else } qsort(less(hd(l), tl(l))) <> [hd(l)] <> qsort(more(hd(l), tl(l))) \end{array}$$

where $less(h,t)$ is a list of members of t less than or equal to h, $more(h,t)$ is a list of members of t strictly more than h and $<>$ is the infix list append function. Recursion analysis would use the definitions of *ord* and *perm* to construct induction rule (2) or (3), whereas to synthesis *qsort* we need something like the following hybrid destructor/constructor style induction rule.

$$\frac{\phi([\,]) \quad \forall h{:}\tau.\forall t{:}list(\tau).\ \phi(less(h,t)) \wedge \phi(more(h,t)) \rightarrow \phi([h|t])}{\forall l{:}list(\tau).\ \phi(l)} \tag{5}$$

6 Rippling and Ripple Analysis

Rippling is a heuristic technique for controlling the proof of the induction conclusion with the aid of the induction hypothesis [Bundy *et al*, 2005]. It works by annotating the differences between the conclusion and hypothesis, and then trying to reduce them using annotated rewrite rules called *wave-rules*. Figures 3 and 5 illustrate the process and give examples of annotation and wave-rules.

Givens:

Initially	**Neutralised**
$ord(qsort(\ less(h,t)\ ))$	$ord(qsort(less(h,t)))$
$ord(qsort(\ more(h,t)\ ))$	$ord(qsort(more(h,t)))$

Goal and Ripple:

$$ord(qsort(\ [h|t]^{\uparrow}\ ))$$

$$ord(\ qsort(\ less(h,t)\ ) <> [h|qsort(\ more(h,t)\ )]^{\uparrow}\ ) \tag{6}$$

$$ord(\ qsort(less(h,t)) <> [h|qsort(more(h,t))]^{\uparrow}\ ) \tag{7}$$

$$ord(qsort(less(h,t))) \wedge ord(qsort(more(h,t)))\ \cdots$$

$$\cdots \wedge ord(qsort(less(h,t))) \underline{\ll} [h] \wedge [h] \ll ord(qsort(more(h,t)))^{\uparrow} \tag{8}$$

$$ord(qsort(less(h,t))) \wedge ord(qsort(more(h,t)))^{\uparrow} \tag{9}$$

$$\top \wedge \top^{\uparrow} \tag{10}$$

Wave Rules:

$$qsort(\ [H|T]^{\uparrow}\ ) \Rightarrow qsort(\ less(H,T)\ ) <> [H|qsort(\ more(H,T)\ )]^{\uparrow} \tag{11}$$

$$ord(\ L <> [X|M]^{\uparrow}\ ) \Rightarrow ord(L) \wedge ord(M) \wedge L \underline{\ll} [X] \wedge [X] \ll M^{\uparrow} \tag{12}$$

This example is taken from the step case of a verification proof of $ord(qsort(l))$ using induction rule (5). The two induction hypotheses, in the cases where $l = less(h,t)$ and $l = more(h,t)$, provide the givens. The induction conclusion is the goal to be proved by rippling. Both the givens and the goal are annotated to indicate where they are similar and where they differ. The annotation consists of hollow grey boxes called wave-fronts*; the holes in these wave-fronts are called* wave-holes*. The expressions in the wave-holes are shared by a given and the goal and the expressions in the grey areas are where they differ. At step (7) it becomes possible to increase the size of the two wave-holes by dropping the wave-fronts in the givens and the innermost wave-fronts in the goal. This is because the similarities between the goal and the givens has increased. We call this process* neutralisation*.*

Rippling proceeds by rewriting the goal with the wave-rules. Wave-rules *are rewrite rules annotated with wave-fronts, which mark the similarities and differences between the left- and right-hand sides of the rules. When applying wave-rules, the wave-annotation must match. Notice that the effect of applying the wave-rules is that the content of the wave-holes increases in size until a copy of a given appears inside them. These givens may then be used to replace the copies with $\top$ in step (10). Wave-rule (11) arises from the recursive definition of qsort and is applied at step (6). Wave-rule (12) is a lemma about ord and is applied at step (8). Note that $L \ll M$ means that every element of list L is less than every element of list M and $L \underline{\ll} M$ that every element L is less than or equal to every element M. At step (9) the wave-front is simplified by applying the two lemmas $qsort(less(h,t)) \underline{\ll} [h]$ and $[h] \ll qsort(more(h,t))$.*

Figure 3: An Example of Rippling

Rippling suggests an alternative to recursion analysis for the construction of an appropriate induction rule. Instead of using recursive definitions to suggest induction terms and variables, we can use wave-rules. We call this technique *ripple analysis*. Ripple analysis conducts a one-step look-ahead into the rippling process and suggests an induction rule that would facilitate rippling by providing an induction term that will match the left-hand-side of the wave-rule. Ripple analysis is illustrated in Figure 4, using the same example that we used for recursion analysis, in order to emphasise the difference. In particular, ripple analysis is able to break-out of the recycling of recursive definitional structure by suggesting induction rules based on derived lemmas rather than recursive definitions.

Conjecture: $\forall l{:}list(\tau).\exists m{:}list(\tau).\ perm(l, m)$

Wave-Rule:

$$perm(\ \boxed{K <> L}^{\uparrow}\ ,\ \boxed{K' <> L'}^{\uparrow}\ ) \quad \Rightarrow \quad \boxed{perm(K, K') \wedge perm(L, L')}^{\uparrow}$$

Constructed Induction Rule:

$$\frac{\phi([\,]) \quad \forall h{:}\tau.\ \phi([h]) \quad \forall l{:}list(\tau).\forall m{:}list(\tau).\ \phi(l) \wedge \phi(m) \rightarrow \phi(\ \boxed{l <> m}^{\uparrow}\ )}{\forall l{:}list(\tau).\ \phi(l)}$$

Suppose induction were applied to the conjecture with induction variable l, which, as the only universally quantified variable, is the only induction variable candidate. The above wave-rule would apply to the induction conclusion if the induction term were $\boxed{l <> m}^{\uparrow}$ *. The above induction rule has thus been constructed to provide just such an induction term. Note that the wave-rule is based on a distributive-law lemma about perm, rather than its recursive definition. Of course, other wave-rules will make other induction rule suggestions, including the wave-rule based on the recursive definition of perm, as in recursion analysis.*

Figure 4: An Example of Ripple Analysis

Unfortunately, ripple analysis does not always suggest the optimal induction rule. The main problem is that it conducts only a *one-step* look-ahead into the rippling process. Later rippling steps may put additional requirements on the induction term that are not apparent at the first step. Figure 5 gives an example of the failure of ripple analysis.

7 Middle-Out Reasoning

In §5 we described the problem of constructing induction rules to prove synthesis theorems. In this section we propose the following solution to this problem. We will use higher-order meta-variables as a *least-commitment device* to postpone the construction of the induction rule. In particular, we will assume that we have a recursive definition for the synthesised program, but use meta-variables to stand for the constructor and destructor functions in its definition. We will then proceed with the synthesis proof. Just as in ripple analysis, we will construct an induction rule that allows rippling to proceed, but this rule will contain meta-variables, so will only be partially defined. During the course of the synthesis proof, higher-order unification will instantiate the meta-variables, firming up both the induction rule and the synthesised program's definition. This will allow not just the first but *all* the ripple steps in the proof to side-effect the construction of the induction rule. This increased flexibility comes at a price: a larger search space. Applying higher-order unification to meta-variables in the goals, increases the branching rate of rippling.

Conjecture:

$$\forall x, y, z{:}\mathbb{N}.\ even(x + y) \wedge even(y + z) \ \rightarrow\ even(x + z)$$

Wave-Rules:

$$s(M)^{\uparrow} + N \quad \Rightarrow \quad s(M + N)^{\uparrow} \tag{13}$$

$$even(\ s(s(X))^{\uparrow}\ ) \quad \Rightarrow \quad even(X) \tag{14}$$

Goal and Ripple:

$$even(\ s(x)^{\uparrow} + y) \wedge even(y + z) \quad \rightarrow \quad even(\ s(x)^{\uparrow} + z)$$

$$\underbrace{even(\ s(x + y)^{\uparrow}\ )}_{\textbf{blocked}} \wedge even(y + z) \quad \rightarrow \quad \underbrace{even(\ s(x + z)^{\uparrow}\ )}_{\textbf{blocked}}$$

One-Step Induction Rule Constructed by Ripple Analysis:

$$\frac{\phi(0) \quad \forall n{:}\mathbb{N}.\ \phi(n) \rightarrow \phi(\ s(n)^{\uparrow}\ )}{\forall n{:}\mathbb{N}.\ \phi(n)} \tag{15}$$

A Better, Two-Step Induction Rule:

$$\frac{\phi(0) \quad \phi(s(0)) \quad \forall n{:}\mathbb{N}.\ \phi(n) \rightarrow \phi(\ s(s(n))^{\uparrow}\ )}{\forall n{:}\mathbb{N}.\ \phi(n)} \tag{16}$$

The rippling analysis one-step look-ahead uses wave-rule (13) to construct the one-step induction rule (15), with x as induction variable. x is preferred to y and z because wave-rule (13) will be able to ripple both occurrences of $s(x)$. If y were chosen, only the second occurrence of $s(y)$ could be rippled and if z were chosen, neither occurrence of $s(z)$ could be rippled. Rippling proceeds in the induction conclusion using the above wave-rule (13) on both sides of the implication, but then gets blocked on both sides of the implication. Ideally, wave-rule (14) would now be applied on both sides, but it requires a wave-front containing two nested occurrences of s, rather than just one. We should have used the two-step induction rule (16), instead of (15).

Figure 5: An Example of a Ripple Analysis Failure

Fortunately, the additional rippling requirement that wave annotation in goal and wave-rule must match, dramatically decreases what would otherwise by an unacceptable combinatorial explosion. We will have a proof obligation to show that the constructed induction rule is valid, i.e. well-founded and covering.

We call this technique *middle-out reasoning*, since it postpones the early proof-search decisions, allowing these to be decided as a side-effect of the proof process in the middle of the proof. Middle-out reasoning is illustrated in Figure 6.

Middle-out reasoning is a simple idea, but has proven to be surprisingly difficult to implement. Accordingly, our first attempt addressed a simplified version of the problem. Instead of dynamically constructing induction rules using a middle-out version of ripple analysis, we merely selected them from a pre-verified store. This work was conducted as a PhD project by Ina Kraan [Kraan *et al*, 1996, Kraan, 1994]. It was implemented in the *Periwinkle* system and applied to the successful synthesis of a number of logic programs. More recently, we have tackled the full problem of middle-out construction of induction rules via the PhD project of Jeremy Gow [Gow, 2004].

Note that middle-out ripple-analysis suggests candidate induction terms as a side-effect of rippling. These must then be turned into a valid induction rule. This may require combining several distinct induction term suggestions into complementary step cases of the same induction rule. It may also require the construction of the corresponding base cases. The induction rule must then be proved to be valid. This involves showing that it is based on a well-founded order and that the base and step cases cover the data-structure. To simplify the well-foundedness proofs, we restricted the well-found measures to those arising in Walther's estimation calculus [Walther, 1994, Gow *et al*, 1999]. This provides a simple family of well-found measures, but with wide coverage, including most practical algorithms.

In practice, the construction of the base and step cases is interleaved with the proofs of coverage and well-foundedness, i.e. base and step cases are invented to fill gaps in the coverage proof, and a well-founded order is evolved to include all these step cases. It is this interaction between the different proof processes that makes middle-out induction-rule construction such a challenging task. The resulting proof obligations are illustrated in Figure 7.

Despite this challenge, middle-out induction-rule construction has been totally automated within the *Dynamis* system, building on the λ*Clam* proof planner, [Div2000]. Dynamis has been successfully applied to the proof of properties of higher-order, functional programs, especially to theorems whose proofs require novel induction rules. Figure 1 shows a selection of theorems proved and the induction rules automatically constructed in those proofs. At the time of writing, Dynamis is being ported to the IsaPlanner proof planner [Dixon & Fleuriot, 2003] and adapted for deductive synthesis.

8 Related Work

The standard technique for choosing induction rules in explicit-induction theorem provers is recursion analysis. We have already compared this to ripple analysis in §6.

Many people have investigated the synthesis of programs from specifications of their required behaviour. Some approaches have used constructive type theory and some have used classical logics. There have been varying degrees of automation, including the use of rippling. However, the discussion below will focus only on mechanisms for the construction of non-standard induction rules and least-commitment mechanisms to support such construction.

8.1 Protzen's Lazy Induction

Apart from the work of Kraan and Gow, described above, we are aware of one other attempt to use least commitment devices to postpone the choice of induction rule and then incrementally build an appropriate induction rule during proof search. This is the work of Protzen on *Lazy Induction* [Protzen, 1995]. It uses a copy of the conjecture as the induction conclusion, proves the step cases

Givens:

Initially	**Neutralised**	**Instantiated**
$ord(qsort(\boxed{D_1(\vec{h},t)}))$	$ord(qsort(D_1(\vec{h},t)))$	$ord(qsort(less(h,t)))$
$\vdots$	$\vdots$	
$ord(qsort(\boxed{D_n(\vec{h},t)}))$	$ord(qsort(D_n(\vec{h},t)))$	$ord(qsort(more(h,t)))$

Goal and Ripple:

$$ord(qsort(\boxed{C(\vec{h},t)}^{\uparrow}))$$

$$ord(\boxed{F(\vec{h},t,qsort(\boxed{D_1(\vec{h},t)}),\ldots,qsort(\boxed{D_n(\vec{h},t)}))}^{\uparrow}) \quad (17)$$

$$ord(\boxed{F(\vec{h},t,\boxed{qsort(D_1(\vec{h},t))},\ldots,\boxed{qsort(D_n(\vec{h},t))})}^{\uparrow}) \quad (18)$$

$$\boxed{ord(qsort(D_1(h,t)))} \wedge \boxed{ord(qsort(D_2(h,t)))} \wedge qsort(D_1(h,t)) \underline{\ll} [h] \wedge [h] \ll qsort(D_2(h,t))^{\uparrow} \quad (19)$$

$$\boxed{ord(qsort(less(h,t)))} \wedge \boxed{ord(qsort(more(h,t)))}^{\uparrow} \quad (20)$$

$$\boxed{\top} \wedge \boxed{\top}^{\uparrow}$$

Wave Rules:

$$qsort(\boxed{C(\vec{H},T)}^{\uparrow}) \Rightarrow \boxed{F(\vec{H},T,qsort(\boxed{D_1(\vec{H},T)}),\ldots,qsort(\boxed{D_n(\vec{H},T)}))}^{\uparrow} \quad (21)$$

$$ord(\boxed{L <> [X|M]}^{\uparrow}) \Rightarrow \boxed{ord(L)} \wedge \boxed{ord(M)} \wedge L \underline{\ll} [X] \wedge [X] \ll M^{\uparrow} \quad (22)$$

This synthesis proof fragment above follows the same pattern as the verification proof in Figure 3 except that meta-variables are used to represent unknown structure. Wave-rule (21) is derived from the initially unknown, schematic definition of qsort. In this schematic definition, we use the second-order meta-variables C and the D_j to represent the initially unknown constructor and destructor functions, and F to represent the body. The dotted boxes around these meta-variables represent potential *wave-fronts — since we are not yet sure which of them will be non-trivial*[1]*. The aim of the synthesis proof is to instantiate these meta-variables and, hence, to construct the definition of qsort. This instantiation will turn wave-rule (21) into wave-rule (11).*

Just as in Figure 3, ripple analysis constructs an induction rule based on the definitional wave-rule (21). This wave-rule is then used to ripple the goal (step (17)). The wave-fronts around the D_j destructor functions in both givens and goal are then neutralised and removed (step (18)). At step (19) wave-rule (22) applies. This application requires second-order unification to instantiate F to $\lambda h.\lambda t.\lambda l.\lambda m.\ l <> [h|m]$. As a result, $\vec{h}$ becomes h and n is instantiated to 2 by the discovery of the arity of F, i.e. the number of recursive calls is established. At step (20) the resulting wave-front is simplified away with the lemmas $qsort(less(h,t)) \underline{\ll} [h]$ and $[h] \ll qsort(more(h,t))$. This time, the second-order unification instantiates D_1 to less and D_2 to more, completing the instantiation of the schematic qsort definition into a concrete one. The remainder of the proof proceeds as in Figure 3.

Figure 6: Middle-Out Synthesis

Conjecture:

$$\forall \circ : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}.\forall x{:}\mathbb{N}.\forall l{:}list(\mathbb{N}).\ foldleft\_tr(\circ, x, l) = foldleft(\circ, x, rev(l))$$

foldleft and foldleft_tr are second-order functions that repeatedly apply their first argument to the elements of the list in their third argument, starting with their second argument. foldleft deals with the elements of the list in first to last order, but foldleft_tr deals with them in reverse order, i.e.

$$foldleft(\circ, x, [e_1, \ldots, e_n]) = (e_n \circ (\ldots (e_1 \circ x) \ldots)) = foldleft\_tr(\circ, x, [e_n, \ldots, e_1])$$

Schematic step case proof:

$$\begin{array}{rcl} foldleft\_tr(\circ, x, \boxed{C}^{\uparrow}) & = & foldleft(\circ, x, rev(\boxed{C}^{\uparrow})) \\ \boxed{C'' \circ foldright\_tr(\circ, x, C')}^{\uparrow} & = & foldleft(\circ, x, rev(\boxed{C' <> [C'']}^{\uparrow}\ D))) \end{array}$$

We show how the induction rule from the foldleft example from figure 1 is constructed. Since l is the only universal variable with a recursively defined type, it is chosen as induction variable and is replaced by a second-order meta-variable C in the induction conclusion. An attempt to ripple foldleft_tr with its definition fails. After backtracking through this failed ripple, it is rippled instead with the following wave-rule:

$$foldleft\_tr(F, X, \boxed{L <> [Y]}^{\uparrow}) \Rightarrow \boxed{F(Y, foldright\_tr(F, X, L))}^{\uparrow}$$

which is based on a lemma. This ripple instantiates C to $C' <> [C'']$ and so introduces the essential structure of the induction term. We can now prove this step case of the induction well-founded. The ripple also turns the potential wave-fronts into concrete wave-fronts. Rippling continues to successful fertilization, but only after some further instantiation in the well-foundedness proof[2].

Show step case well-founded: $C' \prec C' <> [C'']$

The estimation calculus is given the task of finding a well-founded measure $\prec$ to show that the induction term $C' <> [C'']$, suggested by this ripple, is strictly greater than the corresponding term, C', in the induction hypothesis. It succeeds with a measure based on the length of the list.

Discover missing cases: $\forall l{:}list(\mathbb{N}).A \vee \exists e{:}\mathbb{N}, f{:}list(\mathbb{N}).\ l = f <> [e]$

We know one case of the induction rule and need to discover any others. In this case, the meta-variable A will be instantiated to $l = [\,]$, i.e. one base case is discovered. In general, any number of base or step cases might be needed.

Prove new cases: $foldleft\_tr(\circ, x, [\,]) = foldleft(\circ, x, rev([\,]))$

We now need to prove the theorem for the newly discovered base case.

Constructed induction rule:

$$\frac{\phi([\,]) \quad \forall e{:}\tau.\forall f{:}list(\tau).\ \phi(f) \rightarrow \phi(\boxed{f <> [e]}^{\uparrow})}{\forall l{:}list(\tau).\ \phi(l)}$$

Figure 7: Proof Obligations in Middle-Out, Induction-Rule Construction

Theorem	**InductionRule**
$\forall x, y, z{:}\mathbb{N}.$ $even(x+y) \wedge even(y+z) \;\rightarrow\; even(x+z)$	$\dfrac{\phi(0) \quad \phi(s(0)) \quad \forall n{:}\mathbb{N}.\ \phi(n) \rightarrow \phi(\ s(s(\underline{n}))\ ^{\uparrow}\ )}{\forall n{:}\mathbb{N}.\ \phi(n)}$
$\forall \circ : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}. \forall x{:}\mathbb{N}. \forall l{:}list(\mathbb{N}).$ $foldleft\_tr(\circ, x, l) = foldleft(\circ, x, rev(l))$	$\dfrac{\phi([\,]) \quad \forall e{:}\tau. \forall f{:}list(\tau).\ \phi(f) \rightarrow \phi(\ f <> [e]\ ^{\uparrow}\ )}{\forall l{:}list(\tau).\ \phi(l)}$

The first example is similar to that in Figure 5. Middle-out reasoning removes the restriction to just a one-level look-ahead. When the ripple gets to the point where the definition of even should be applied, there is still an uninstantiated meta-variable that can be used to influence the induction term and produce the two-step induction rule required. The second example is explained in more detail in Figure 7. It shows that the middle-out reasoning can be applied to higher-order theorems and can construct quite ad hoc *induction rules.*

Table 1: Selected Experimental Results of the Dynamis System

using rippling, invents new induction hypotheses as needed for fertilization and then shows that the emerging induction rule is well-founded.

Lazy Induction has three disadvantages over middle-out induction-rule construction and no advantages of which we are aware.

1. It is restricted to destructor-style, whereas Dynamis can destructor, constructor and hybrid styles.
2. It lacks search control mechanisms to deal with the inherent threat of non-termination when potential wave-fronts are used (see Figure 7), whereas Dynamis has mechanisms for preventing non-termination.
3. It can mix up induction rules in its search, leading to the wasted exploration of impossible combinations of step cases, whereas Dynamis only explores compatible step cases. For instance, suppose two different induction rules each have two step cases. Lazy Induction will explore all four combinations of step case. Dynamis uses shared meta-variables to restrict the search to the two compatible combinations. When a meta-variable is instantiated by proof search in one case it is automatically instantiated to the same value in the other case, which will rule out incompatible combinations.

8.2 Hutter's Labelled Fragments

Hutter has also used rippling as the basis of an attempt to construct induction rules for synthesis conjectures [Hutter, 1994]. His approach is also based on rippling and proof planning. Before attempted the concrete proof, his INKA system first builds an abstract plan of the inductive step cases. The induction rule is constructed from this plan. To form the plan, he uses abstractions of wave-rules, in which the details of the wave-fronts are removed, leaving only the information that it is possible to ripple past skeleton fragments. The induction rule is then constructed by recovering and combining the abstracted wave-fronts. Although Hutter's technique does not use higher-order unification on meta-variables, as Kraan's, Protzen's and Gow's do, it has a similar effect. He has successfully applied it to the synthesis of the inverses of standard functions, using these functions as the specifications, i.e. proving $\forall i. \exists o.\ f(o) = i$ to synthesise an inverse of f. Amongst the functions synthesised are log, $half$, *quotient* and rev. The synthesis of rev, for instance, constructs induction rule (4).

8.3 Narrowing

Narrowing [Hanus, 1994] is the extension of rewriting to allow the *unification* of the rewrite rule with the goal, rather than just matching, i.e. the goal can contain meta-variables, which may be instantiated during rewriting. Our middle-out reasoning is essentially the extension of rippling to narrowing. The main role of narrowing has been to the construction of existential witnesses. On negation of the conjecture and skolemization[3], existential witness become meta-variables. Narrowing will instantiate these meta-variables as a side-effect of rewriting, incrementally constructing the existential witness.

The main difference between Dynamis and previous work on narrowing is that we are using meta-variables to stand for unknown induction terms as well as existential variables. In other work we have also used meta-variables to stand for unknown structure in generalised conjectures and in missing lemmas [Ireland & Bundy, 1996].

9 Conclusion

In this paper we have discussed the issue of constructing an appropriate induction rule for the automated synthesis of computer programs by constructive proof. Induction is a vital ingredient of any synthesis proof of a program containing iteration or recursion. There are an infinite number of induction rules for each non-trivial data-type, so induction rules need to constructed to order rather than pre-stored. However, recursion analysis, the standard technique for induction rule construction, was developed for purely universally quantified theorems. Synthesis conjectures always contain existential quantifiers. Applied to synthesis theorems, recursion analysis merely adapts the recursions in the specification of a program. It would be incapable of suggesting a novel recursive structure, such as that used in the quick-sort algorithm, for instance.

We have developed a series of techniques for induction-rule construction that are not limited to recycling the recursions in the original conjecture. These are based on a combination of ripple analysis and middle-out reasoning. They have been successfully used to construct novel induction rules automatically. Two principles are at work.

1. Induction terms and hence induction rules are suggested by applying lemmas and not just recursive definitions.

2. Constructing the induction term, and hence the induction rule, is postponed by the use of meta-variables. These are incrementally instantiated during the course of the proof, so that the requirements of several different proof steps can be taken into account in the shape of the constructed induction rule.

The least commitment mechanisms used in middle-out induction-rule construction lead to a complex juggling of proof obligations. Not only must the original theorem be proved, but the induction rule must be shown both to be well-founded and to cover the data-type. As a side effect of these proof obligations, the following objects are constructed: induction terms, a well-founded measure, missing base and step cases and an induction rule. Each proof obligation is sharing and instantiating a set of meta-variables. It is necessary to co-routine between these proofs: as the instantiation of a meta-variable in one proof obligation sufficiently restricts proof search in another, temporarily frozen, proof obligation to allow it to restart safely.

Initial experiments with middle-out induction-rule construction have been limited to purely universal theorems, such as may arise in verification proofs. We are now turning our attention to theorems containing existential quantifiers, as required in synthesis proofs. The Dynamis system is being ported to the IsaPlanner proof planner and applied to the synthesis of programs from their specifications. Proof planning has played an essential role in this work, for instance, enabling the flexible construction of proofs using middle-out reasoning and providing the powerful rippling method.

[3] Or, equivalently, dual-skolemization of the goal.

In addition to applying our techniques to a growing corpus of synthesis conjectures, we plan to extend our approach to cope with having an unknown number of recursive calls in a schematic, recursive definition. We also want to allow more flexible co-routining between different proof obligations. For instance, we want to be able to freeze partial proofs that have too high a branching rate and to unfreeze partial proofs whose branching rates have been significantly reduced as a side-effect of shared meta-variable instantiation in other proofs since they were previously frozen.

References

[Boyer & Moore, 1979] Boyer, R. S. and Moore, J S. (1979). *A Computational Logic*. Academic Press, ACM monograph series.

[Bundy *et al*, 2005] Bundy, A., Basin, D., Hutter, D. and Ireland, A. (2005). *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press.

[Darlington, 1978] Darlington, J. (1978). A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30.

[Div2000] Division of Informatics, University of Edinburgh, Edinburgh. (v2000). *User/Programmer Manual for the λClam proof planner*, v2.0.0 edition.

[Dixon & Fleuriot, 2003] Dixon, L. and Fleuriot, J. D. (2003). IsaPlanner: A prototype proof planner in Isabelle. In *Proceedings of CADE'03*, Lecture Notes in Computer Science.

[Gow, 2004] Gow, Jeremy. (2004). *The Dynamic Creation of Induction Rules Using Proof Planning*. Unpublished Ph.D. thesis, School of Informatics, University of Edinburgh.

[Gow *et al*, 1999] Gow, J., Bundy, A. and Green, I. (1999). Extensions to the estimation calculus. In Ganzinger, H., McAllester, D. and Voronkov, A., (eds.), *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, pages 258–272, Tblisi, Georgia. Springer–Verlag. LNAI 1705.

[Hanus, 1994] Hanus, M. (1994). The integration of functions into logic programming: from theory to practice. *J. Logic Programming*, 19 & 20:583–628.

[Hutter, 1994] Hutter, D. (1994). Synthesis of induction orderings for existence proofs. In Bundy, Alan, (ed.), *12th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 814, pages 29–41, Nancy, France. Springer-Verlag.

[Ireland & Bundy, 1996] Ireland, A. and Bundy, A. (1996). Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111. Also available from Edinburgh as DAI Research Paper No 716.

[Kraan, 1994] Kraan, I. (1994). *Proof Planning for Logic Program Synthesis*. Unpublished Ph.D. thesis, Department of Artificial Intelligence, University of Edinburgh.

[Kraan *et al*, 1996] Kraan, I., Basin, D. and Bundy, A. (1996). Middle-out reasoning for synthesis and induction. *Journal of Automated Reasoning*, 16(1–2):113–145. Also available from Edinburgh as DAI Research Paper 729.

[Protzen, 1995] Protzen, M. (February 1995). *Lazy Generation of Induction Hypotheses and Patching Faulty Conjectures*. Unpublished Ph.D. thesis, Technische Hochschule Darmstadt, Darmstadt, Germany.

[Walther, 1993] Walther, C. (1993). Combining induction axioms by machine. In *Proceedings of IJCAI-93*, pages 95–101. International Joint Conference on Artificial Intelligence.

[Walther, 1994] Walther, C. (1994). On proving termination of algorithms by machine. *Artificial Intelligence*, 71(1):101–157.